🎯 Grid Path Finder

Objective:

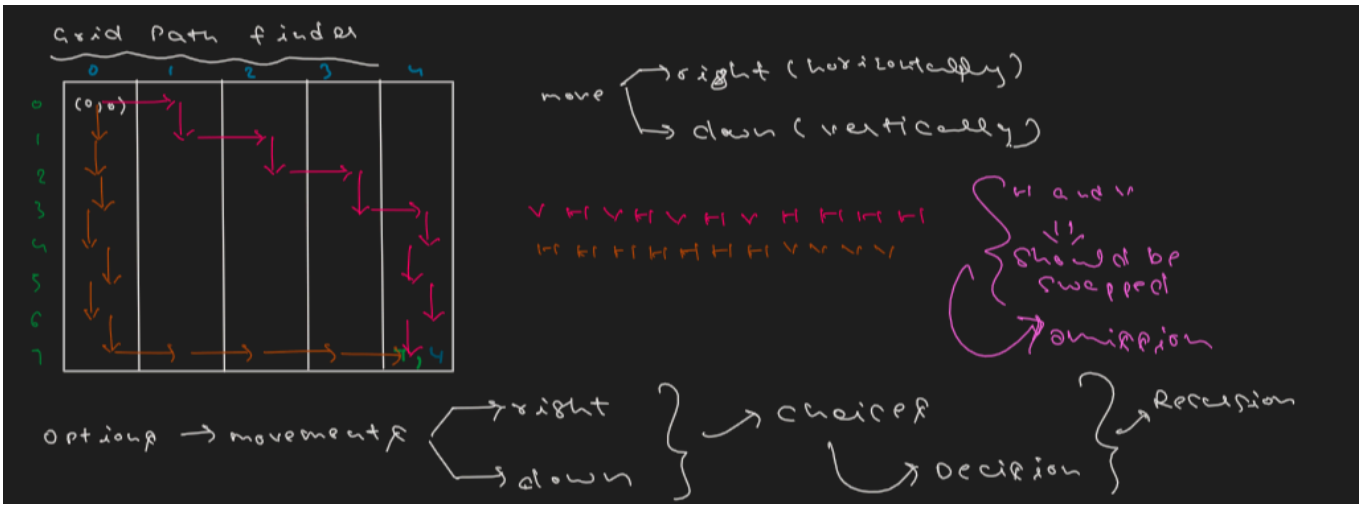
Given a 2D grid of size `n x m`, starting from the top-left corner `(0,0)`, the goal is to print all possible paths to reach the bottom-right corner `(n-1, m-1)` using only the following two moves:

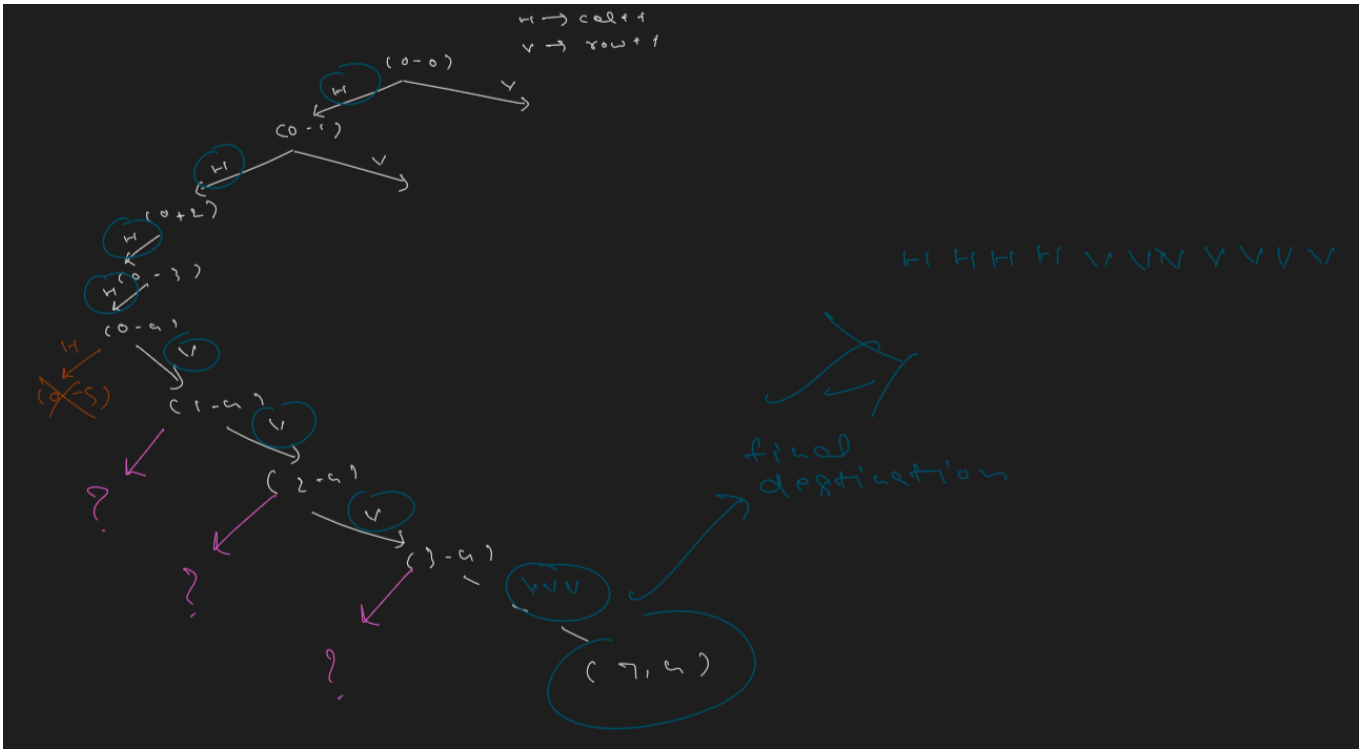
- Horizontal move (H) → move right by one column.
- Vertical move (V) → move down by one row.

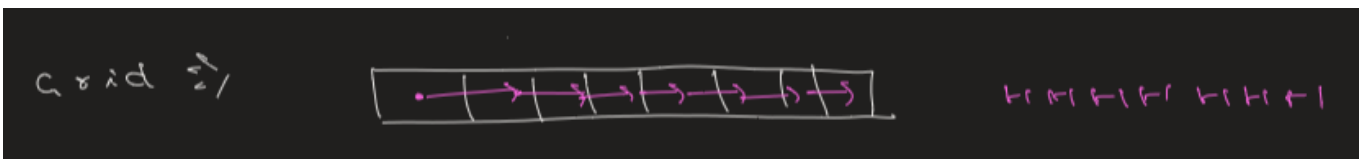
Each path should be printed in the order of moves taken, ending with the word "STOP" to indicate the destination has been reached.

Constraints:

- You can only move right or down at any point in time.
- You cannot move out of the grid.
- All paths from `(0, 0)` to `(n-1, m-1)` must be explored and printed.

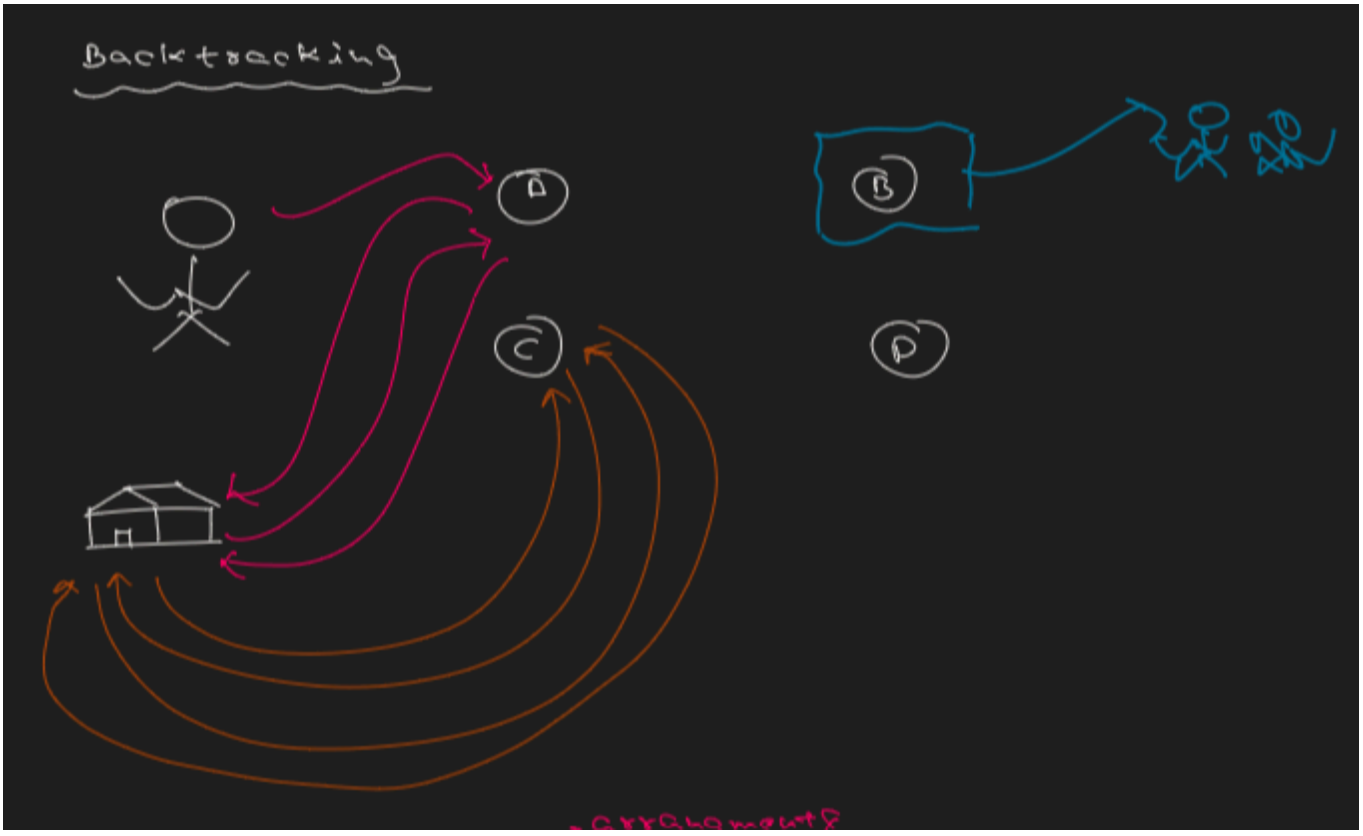






```java
public class Main {
	public static void main(String[] args) {
		int n = 3;
		int m = 4;
		path(0, 0, n - 1, m - 1, "");
	}
	// cr--> current row, cc--> current col, er--> end row, ec--> end col

	public static void path(int cr, int cc, int er, int ec, String ans) {
		if (cr == er && cc == ec) {
			System.out.println(ans+ "STOP");
			return;
		}
		if (cr > er || cc > ec) {
			return;
		}
		path(cr, cc + 1, er, ec, ans + "H -> ");
		path(cr + 1, cc, er, ec, ans + "V -> ");
	}

}
```

📌 What is Backtracking?



📘 Backtracking Explained Through a Real-Life Analogy

🧵 Problem Scenario

- A person is trying to meet a girl at NSP, but due to network issues, they cannot contact each other.
- The girl is waiting at one of four restaurants: A, B, C, or D.
- Each restaurant has a specific dress code (constraint).
- The person must start from home, fulfill the dress code, and check each restaurant one by one.

🧭 Step-by-Step Process (Analogous to Backtracking)

🔹 **Restaurant A:**

- Visited A from home.
- Entry denied: Cap required.
- Returned home, wore a cap, and tried again.
- Girl was not there.
- Returned home and removed the cap (undo step).

🔹 **Restaurant B:**

- Visited B from home.
- Entry denied: Black shoes required.
- Returned home, wore black shoes, and tried again.
- Girl was not there.
- Returned home and removed the shoes (undo step).

🔹 **Restaurant C:**

- Visited C from home.
- Entry denied: Sunglasses required.
- Returned home, wore sunglasses, and tried again.
- Girl was not there.
- Returned home and removed sunglasses (undo step).

🔹 **Restaurant D:**

- Visited D from home.
- Entry denied: Formal shirt required.
- Returned home, wore a formal shirt, and tried again.
- ✅ Girl found at Restaurant D.

🎯 Key Concepts Mapped to Backtracking

Real-life Step	Backtracking Concept
Choosing a restaurant	Exploring a decision branch
Dress code requirements	Applying constraints
Returning home	Backtracking to a previous state
Changing outfit	Making a new decision
Finding the girl	Reaching a valid solution

Summary

- Backtracking is a method of solving problems by trying all possible options, and undoing decisions when constraints are not met or the solution is not found.

- At each step, it:
    1. Makes a choice.
    2. Checks if the choice leads to a solution.
    3. If not, reverts the choice and tries the next option.
- This process continues until a valid solution is found or all options are exhausted.

🧠 Core Idea

Backtracking is a general algorithmic technique that incrementally builds solutions to a problem and abandons ("backtracks") a solution as soon as it determines that the solution cannot be completed to a valid one.

Backtracking tries to solve a problem by:

1. **Choosing** a possible option.
2. **Exploring** further by making a recursive call.
3. **Validating** if the current choice leads to a solution.
4. **Backtracking** (i.e., undoing the last choice) if the current path doesn't lead to a solution.

This is also called a **depth-first search (DFS)** approach with pruning.

🗒️ Queen *Permutations* on a 1D Board

Objective:

You are given a 1-dimensional board with `n` boxes. Your task is to place `tq` queens on this board in such a way that:

- Each queen is placed in a **different box** (no two queens can occupy the same box).
- The **order** in which the queens are placed matters — different orders are considered different arrangements.
- You must generate and print **all possible valid permutations** of placing `tq` queens on the board.

Each arrangement should be printed in the format:

- `bX qY`
  where `bX` indicates the box index, and `qY` indicates the queen number based on the order of placement (starting from `q0` up to `q(tq-1)`).

Input:

- An integer `n`: the number of boxes on the board.
- An integer `tq`: the total number of queens to place.

Output:

- Print all valid permutations of placing `tq` queens in `n` boxes.
- Each permutation should be printed on a new line.
- Within each line, the order of placements should reflect the order in which the queens are placed.

Example:

Input:

```
n = 4
tq = 2
```

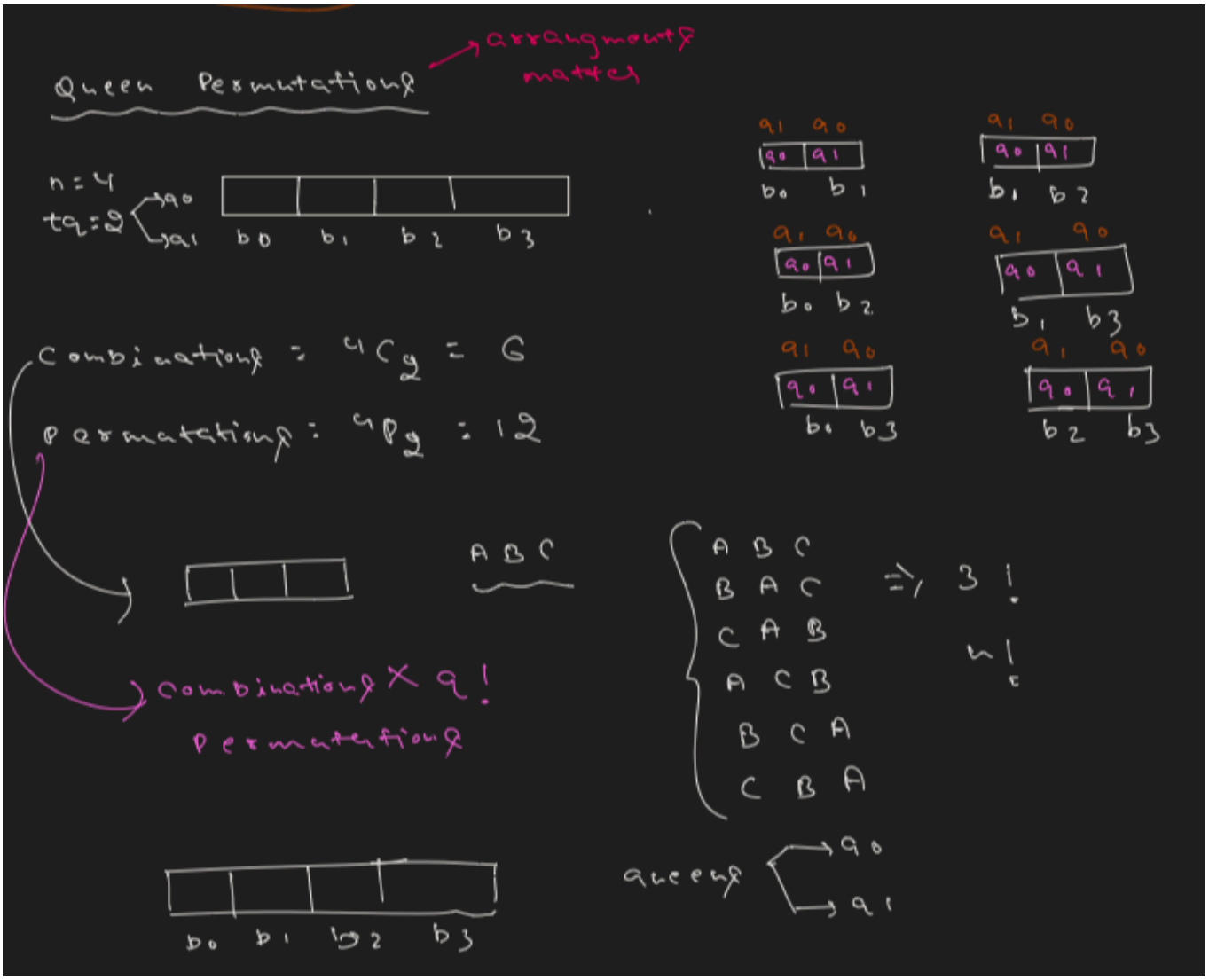
Output:

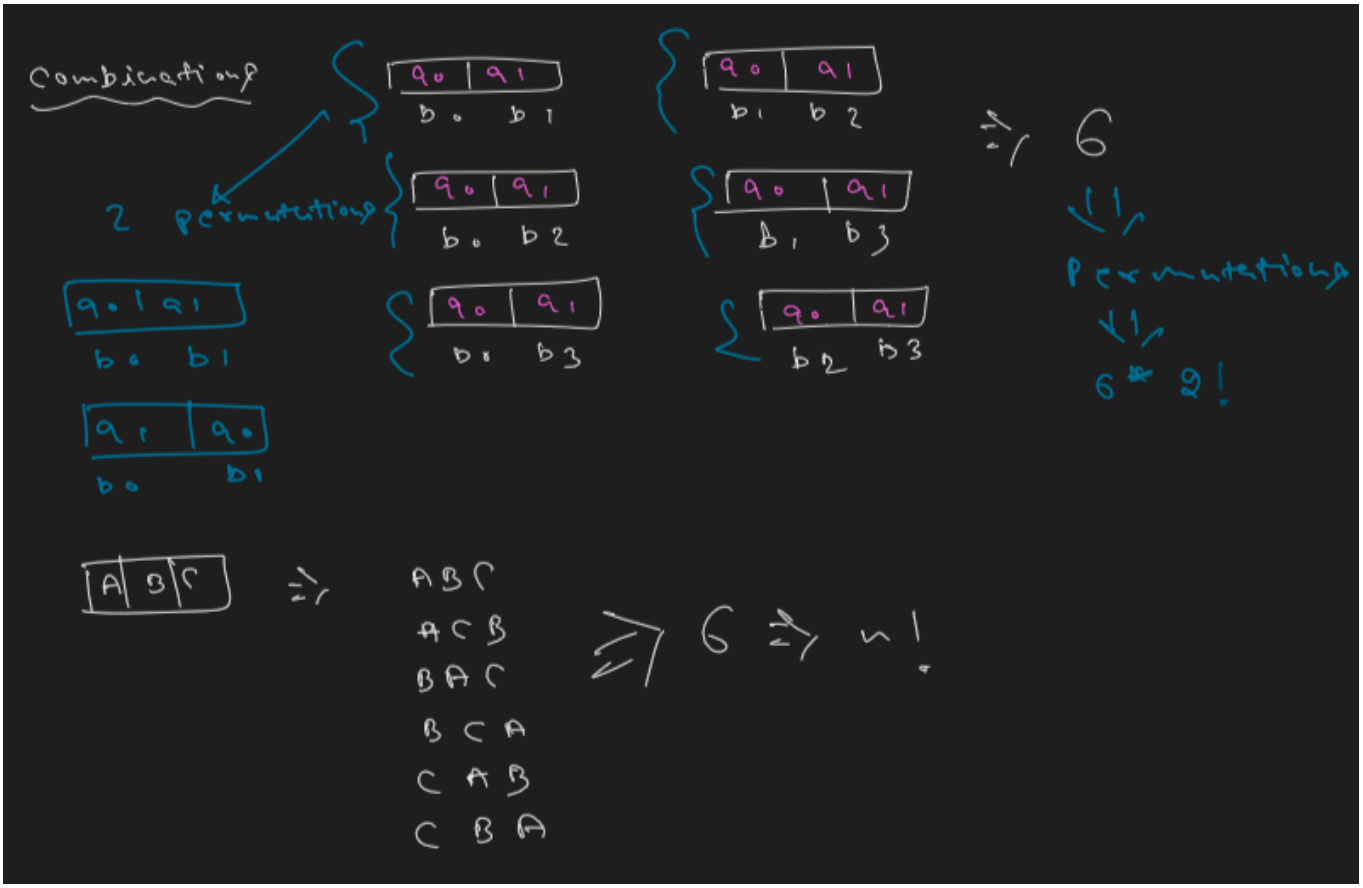
```
b0 q0 b1 q1
b0 q0 b2 q1
```

```
b0 q0 b3 q1
b1 q0 b0 q1
b1 q0 b2 q1
b1 q0 b3 q1
b2 q0 b0 q1
b2 q0 b1 q1
b2 q0 b3 q1
b3 q0 b0 q1
b3 q0 b1 q1
b3 q0 b2 q1
```

Explanation:

In the example, you are placing 2 queens on a board with 4 boxes. All arrangements where queens are placed in different boxes and in different orders are considered valid. There are 12 such permutations in total.





If you look at the problem, the number of **permutations** will be 12, but the number of **combinations** will be only 6 — that is, **4C2**.

This is because to calculate permutations, we consider all possible **orderings** for each combination. So, for every unique combination of two boxes, there are two different ways to assign the two queens (since the order matters).

Basically, you can place the queens in:

- 6 unique **combinations** of two boxes,
- and for each combination, there are 2 **permutations** (queen 0 in box A and queen 1 in box B, or the reverse).

Hence, the total number of permutations is:
4C2 × 2! = 6 × 2 = 12

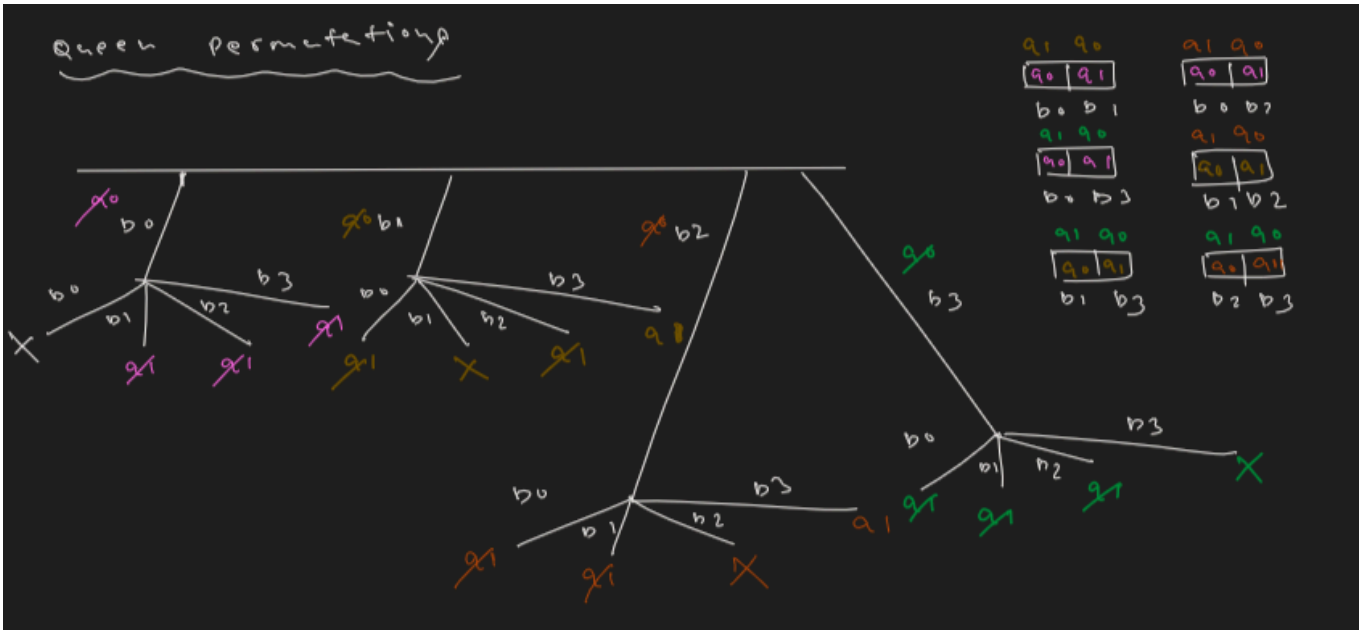
✅ How to Find the Count

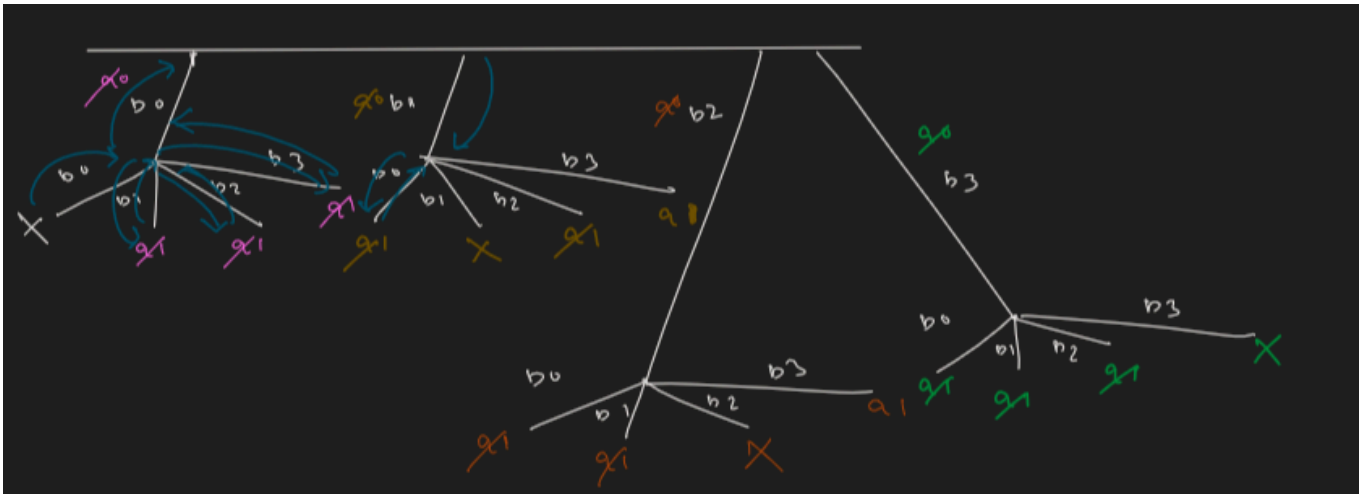
1. First, choose the boxes where the queens will go.
   This is a combination: `nCk` (choose k boxes out of n).
2. Then, arrange the queens in those chosen boxes.
   This is a permutation: `k!` (since order matters)

So the total number of ways = `nCk × k!`

In your case (placing 2 queens in 4 boxes):
4C2 = 6 (ways to choose boxes)
2! = 2 (ways to arrange the queens)

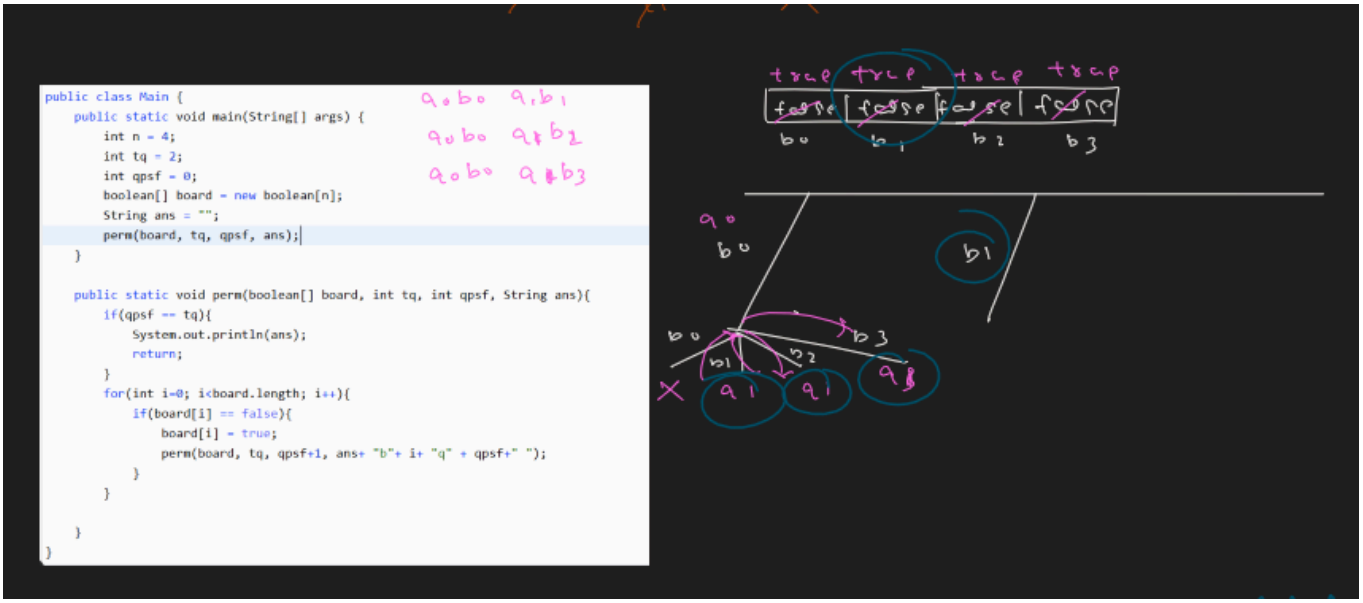
Total permutations = 6 × 2 = **12**.

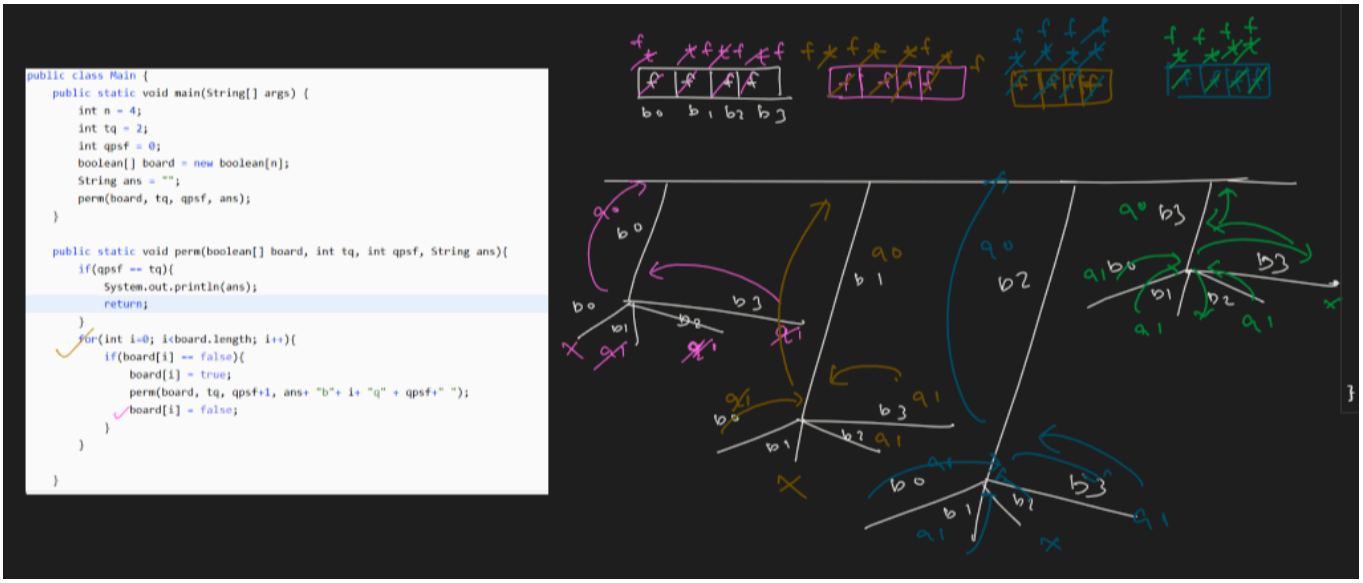




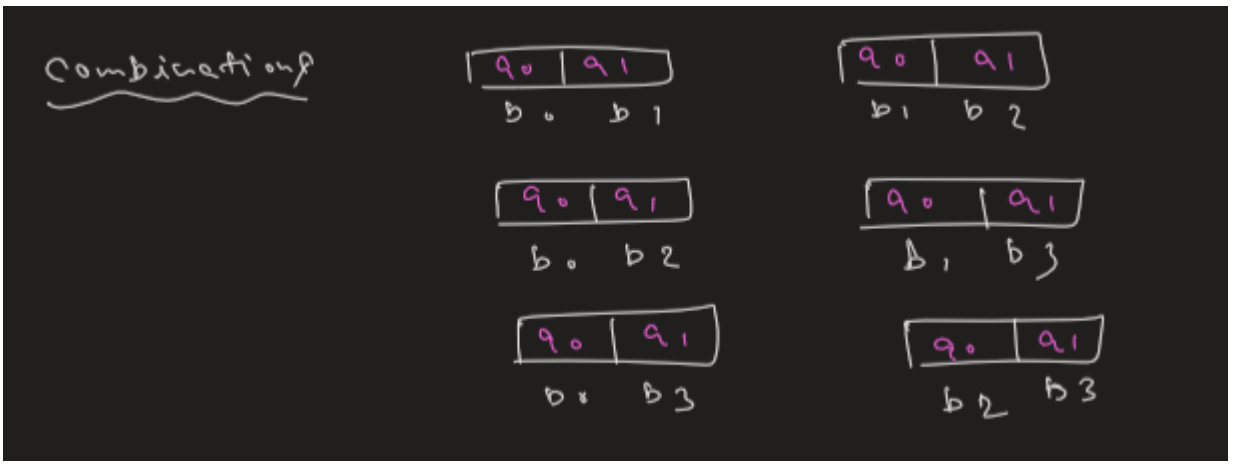
```java
public class Main {
	public static void main(String[] args) {
		int n = 4;
		boolean[] board = new boolean[n];
		int tq = 2; // total queen
		perm(board, tq, 0, "");
	}

	//qpsf -> queen placed so far

	public static void perm(boolean[] board, int tq, int qpsf, String ans) {
		if (qpsf == tq) {
			System.out.println(ans);
			return;
		}
		for (int i = 0; i < board.length; i++) {
			if (board[i] == false) {
				board[i] = true; // queen placed
				perm(board, tq, qpsf + 1, ans + " b" + i + " q" + qpsf);
				board[i] = false; // queen removed -> backtracking
				//DRY RUN before doing backtracking
			}
		}
	}
```

```
}
```



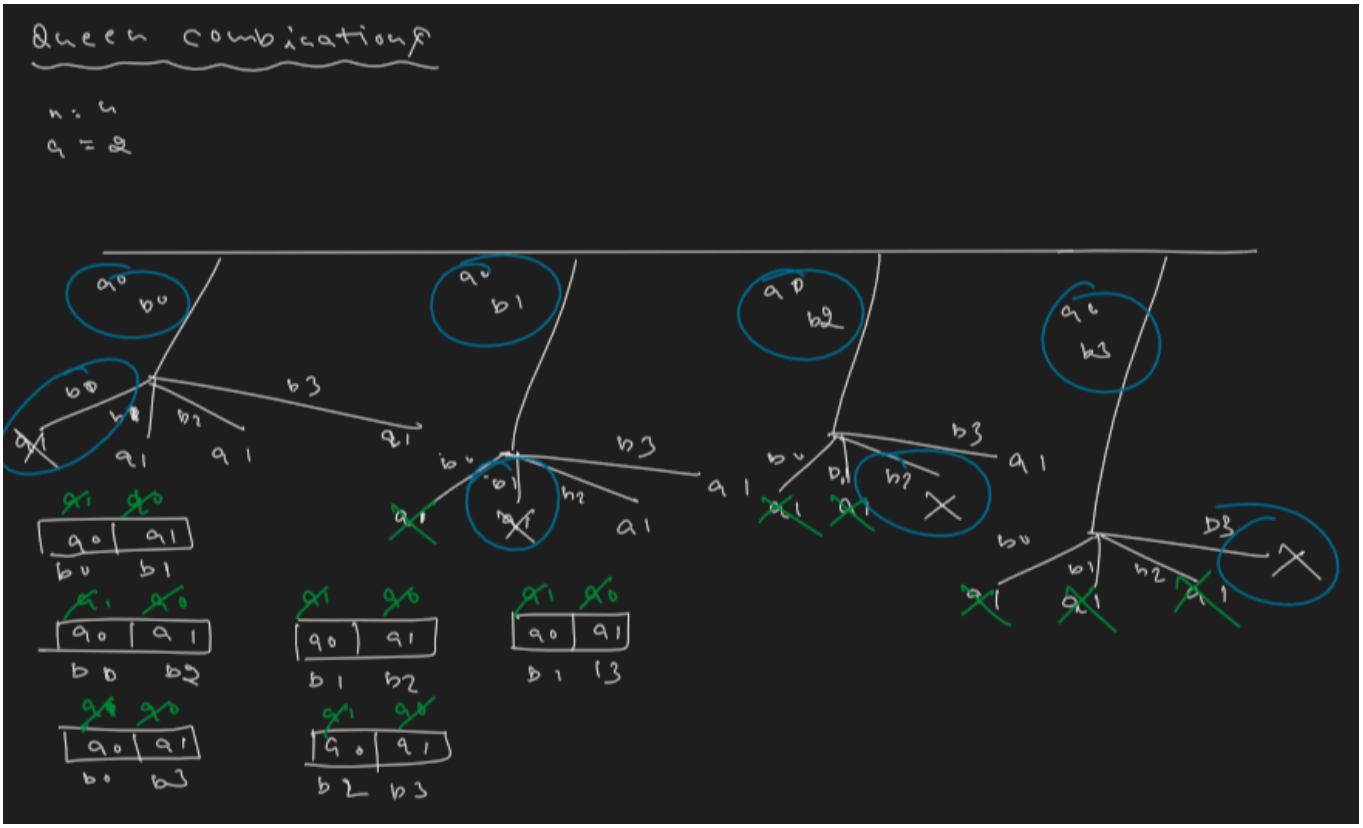


📃 Queen *Combinations* on a 1D Board



Starts from index `idx` and tries placing a queen in each available position (`false`).

Once a queen is placed, recursively tries placing the next queen starting from the next index (`i + 1`) — this avoids repeating the same set in different orders.




```java
public class Main {
	public static void main(String[] args) {
		int n = 4;
		boolean[] board = new boolean[n];
		int tq = 2; // total queen
		com(board, tq, 0, "", 0);
	}

	public static void com(boolean[] b, int tq, int qpsf, String ans, int idx) {
		if (qpsf == tq) {
			System.out.println(ans);
			return;
		}
		for (int i = idx; i < b.length; i++) {
			if (b[i] == false) {
				b[i] = true; // queen placed
				com(b, tq, qpsf + 1, ans + "b" + i + "q" + qpsf, i + 1);
				b[i] = false; // queen removed -> backtracking
			}
		}

	}

}
```

🗒️ Coin Change - *Permutations*

Given an array of coin denominations and a target amount, print **all possible permutations** of coins that sum up to the target amount.

- **You may use each coin an unlimited number of times.**
- **The order of coins matters** (i.e., permutations, not combinations).
- Print each valid sequence as a string of coin values.

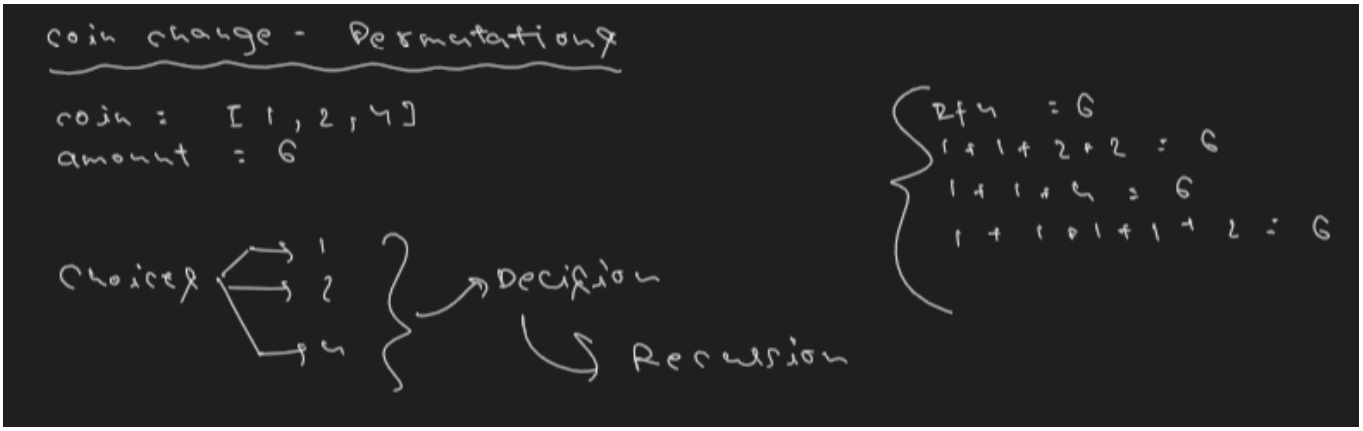
Example:

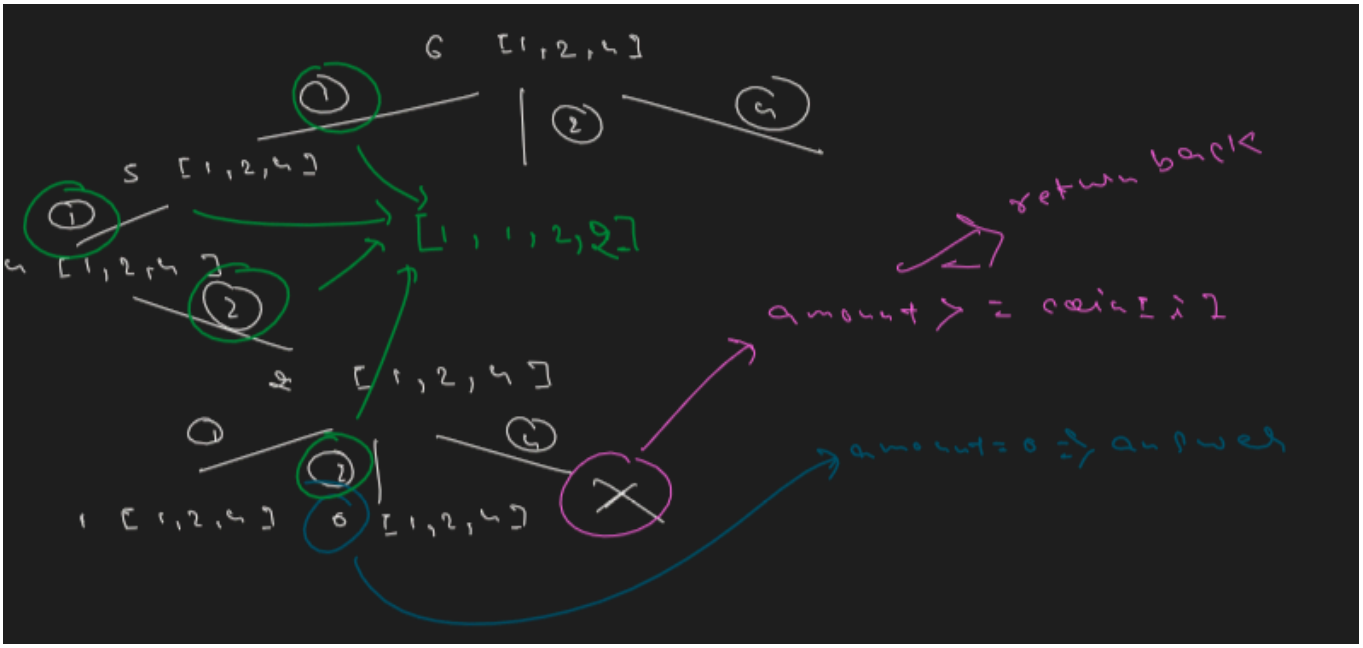
Input:

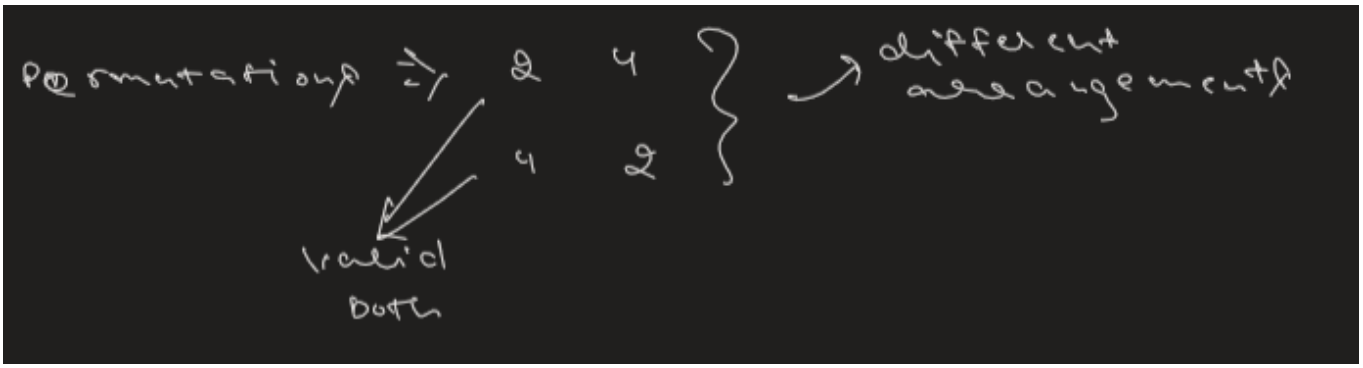
```
coin[] = {1, 2, 4}
amount = 6
```

Output:

```
111111
11112
11121
11211
12111
21111
114
141
411
123
132
213
231
312
321
222
24
42
```








```java
public class Main {
	public static void main(String[] args) {
		int[] coin = { 1, 2, 4 };
		int amount = 6;
		permutation(coin, amount, "");
	}

	public static void permutation(int[] coin, int amount, String ans) {
		if (amount == 0) {
			System.out.println(ans);
			return;
		}

		for (int i = 0; i < coin.length; i++) {
			if (amount >= coin[i]) {
				permutation(coin, amount - coin[i], ans + coin[i]);
			}
		}
	}

}
```

🗒️ Coin Change - *Combinations*

Given an array of coin denominations and a target amount, print all possible **combinations** of coins that sum up to the target amount.

- You may use each coin an **unlimited number of times**.
- The **order of coins does NOT matter** (i.e., combinations, not permutations).
- Print each valid sequence as a string of coin values in **non-decreasing order**.

Example:

Input:

```
coin[] = {1, 2, 4}
amount = 6
```

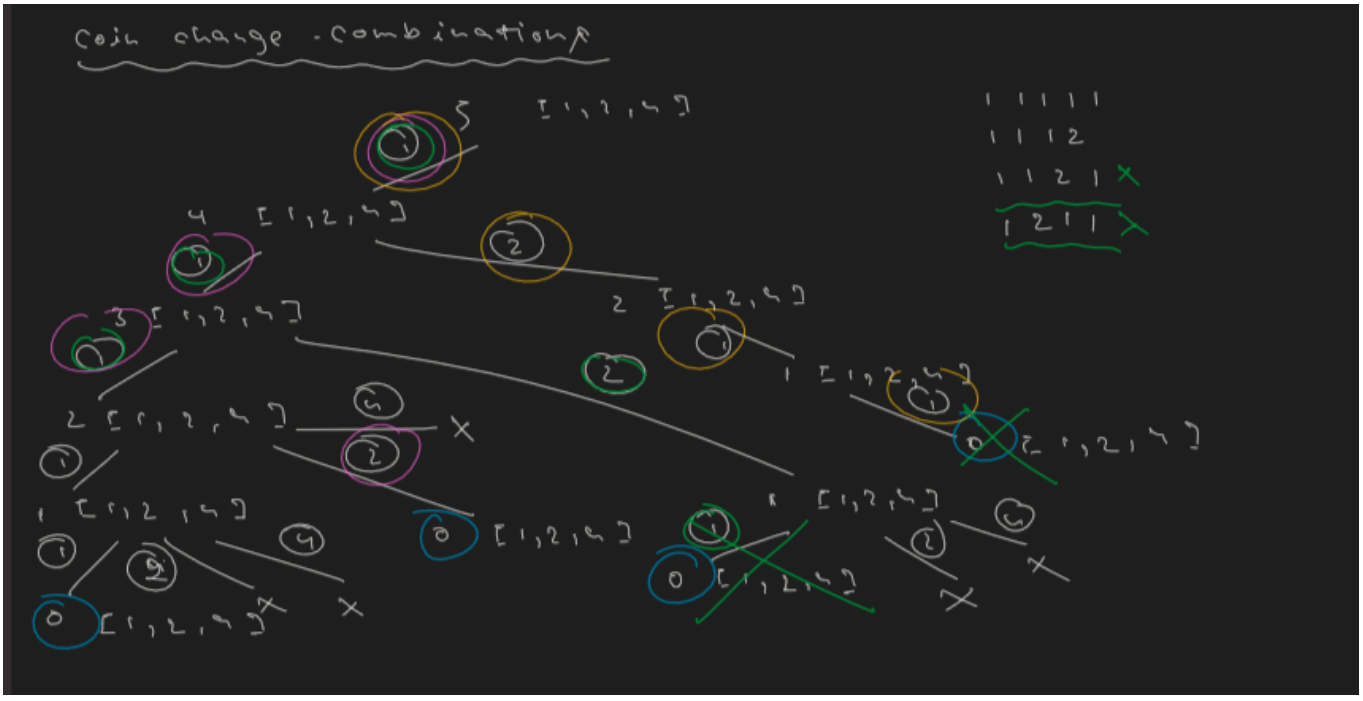
Output:

```
111111
11112
1113
1122
114
222
24
42
```

Note: Outputs like `21111` or `123` are **not printed**, because they are permutations of combinations already printed.

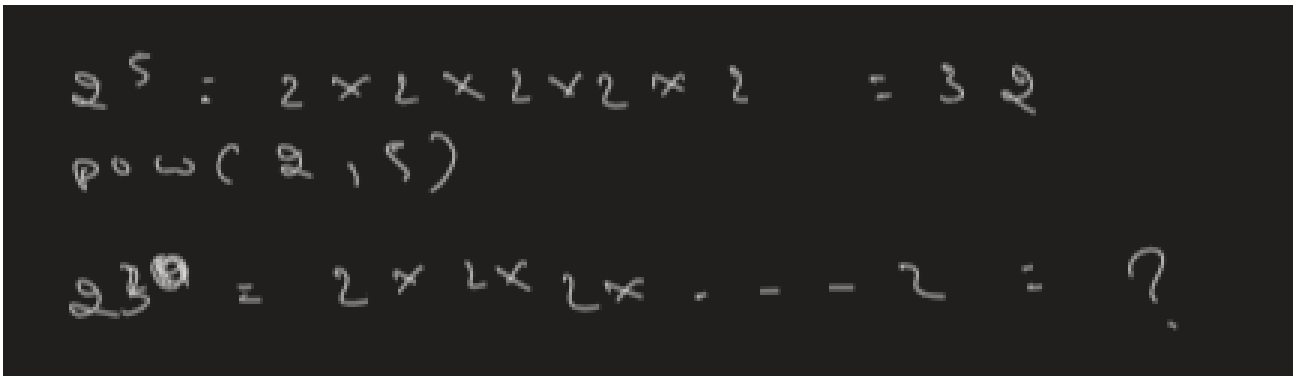
🔁 Explanation:

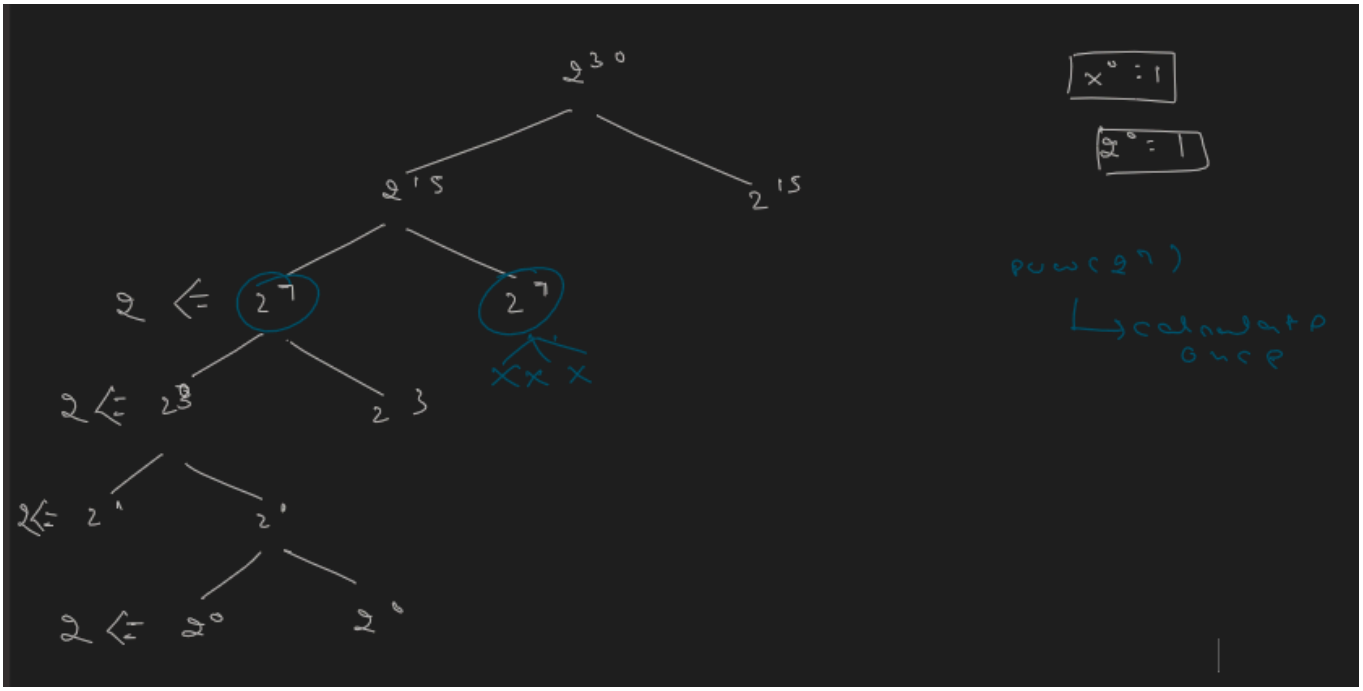
- At each level, we start the loop from `idx` instead of 0 to ensure:
  - Coins are always added in **non-decreasing order**
  - We **avoid generating permutations**
- The same coin can be reused by **not incrementing the index** in the recursive call.



```java
public class Main {
	public static void main(String[] args) {
		int[] coin = { 1, 2, 4 };
		int amount = 6;
		comb(coin, amount, "", 0);
	}

	public static void comb(int[] coin, int amount, String ans, int idx) {
		if (amount == 0) {
			System.out.println(ans);
			return;
		}

		for (int i = idx; i < coin.length; i++) {
			if (amount >= coin[i]) {
				comb(coin, amount - coin[i], ans + coin[i], i);
			}
		}
	}
}
```

🎯 Power function





```java
public class Main {
    public static void main(String[] args) {
        int a = 2;
        int b = 10;
        System.out.println(pow(a, b));
        System.out.println(pow1(a, b));
    }

    public static int pow(int a, int b) {
        if (b == 0) {
            return 1;
        }
        if (b % 2 == 0) {
            return pow(a, b / 2) * pow(a, b / 2);
        } else {
            return 2 * pow(a, b / 2) * pow(a, b / 2);
        }
    }

    public static int pow1(int a, int b) {
        if (b == 0) {
            return 1;
        }
        int half = pow1(a, b / 2);
        if (b % 2 == 0) {
            return half * half;
        } else {
            return 2 * half * half;
        }
    }
}
```